|  |
| --- |
| **Building/Location Equipment Map**  **Final Project Report**  **Final Requirements, Design, Implementation/Testing & Installation/Delivery**  **Allied Telesis**  **CSC 492 Team 10**  **Cameron Lanier**  **Harsh Singh**  **Balaji Sundaram**  **Qiufeng Yu**  **North Carolina State University**  **Department of Computer Science**  **April 24, 2016** |

**Table of Contents**

[**Executive Summary** 3](#_Toc449949260)

[**Project Description** 4](#_Toc449949261)

[1. Sponsor Background 4](#_Toc449949262)

[2. Problem Statement 4](#_Toc449949263)

[3. Project Goals & Benefits 5](#_Toc449949264)

[4. Development Methodology 5](#_Toc449949265)

[5. Challenges 5](#_Toc449949266)

[**Resources Needed** 6](#_Toc449949267)

[**System Requirements** 6](#_Toc449949268)

[Functional Requirements 7](#_Toc449949269)

[Non Functional Requirements 7](#_Toc449949270)

[Constraints 8](#_Toc449949271)

[**System Design** 8](#_Toc449949272)

[High-Level Design 8](#_Toc449949273)

[Backend Design 9](#_Toc449949274)

[Low Level Design 11](#_Toc449949275)

[**Implementation** 17](#_Toc449949276)

[**Testing** 23](#_Toc449949277)

[**Suggestions for Future Teams** 27](#_Toc449949278)

[**Task Plan** 28](#_Toc449949279)

[**Developer’s Guide** 29](#_Toc449949280)

[1. Front End Web Application 30](#_Toc449949281)

[2. Backend Java Application 31](#_Toc449949282)

[**User’s Guide** 33](#_Toc449949283)

# **Executive Summary**

*Author(s): Cameron Lanier, Harsh Singh, Balaji Sundaram (minor edits)*

The Building/Location Equipment Map is a web application based on the AlliedViewTM Network Monitoring System (NMS) software, developed by Allied Telesis. The NMS is used by data service providers to monitor the status of different networking devices and to manage subscribers’ home services. Their current implementation of the network map is a Java application, which has to be installed on a Microsoft Windows device and configured heavily before use. The goal of our project is to create a web based client application that graphically represents the location and status of network equipment in a building, by extending the monitoring functionality of AlliedViewTM. The user experience of the web application allows for a clear representation of the state of devices in the system.

After successful meetings with sponsors, we were able to identify and define the project’s requirements. From the requirements, we were able to create several GUI mockups and design diagrams for the entire system. The main goal of the project was to enable users to place different network devices on a blueprint of a building and view their status. We also understood that the project needs to be expandable so that it can be built upon in the future.

The technologies and components that make up the system include HTML/CSS/JavaScript. We are also using Leaflet and jQuery, which are two JavaScript libraries that help with displaying the blueprints, interacting with the map, and handling events. The user interface communicates to the server via a REST API, which helps both load and save completed layouts and blueprints.

For the remainder of this report, we will provide a detailed description of the project. The report discusses the system requirements and design requirements, using detailed diagrams. These diagrams show a high level design, as well as the user interface for the system. The report also discusses acceptance and unit tests that will help create robust code.

# **Project Description**

*Author(s): Harsh Singh, Balaji Sundaram, Qiufeng Yu, Cameron Lanier (minor edits)*

## 1. Sponsor Background

Allied Telesis is a leader in delivering IP/Ethernet network solutions all over the world. It creates standards-based IP networks that connect customers with voice, video and data services. It was founded in 1987 with the global headquarters in Japan, and the North American headquarters in San Jose, CA. The team we are working with, is located on Centennial Campus and are responsible for creating the AlliedViewTM NMS (Network Monitoring System). This application is used by different internet service providers to manage their subscribers’ network devices. The use of an NMS is becoming increasingly popular, since it can be used to configure and manage routers, wireless access points and switches to monitor the data access of businesses.

Sponsor Contacts

|  |  |
| --- | --- |
| Alasdair McGregor  900 Main Campus Drive  Suite 301  [alasdair\_mcgregor@alliedtelesis.com](mailto:alasdair_mcgregor@alliedtelesis.com) | Jesse Rodman  900 Main Campus Drive  Suite 301  [jesse\_rodman@alliedtelesis.com](mailto:jesse_rodman@alliedtelesis.com) |

## 2. Problem Statement

There is demand for having a graphical representation of the location and status of network equipment by network administrators and network service providers. The existing graphical representation of the NMS is a Java application, which is about 10 years old. Currently, the AlliedViewTM NMS only displays a network map view, which limits the functionality of the application. This view only allows a very basic visual representation with no bearings on the location of any device. In the current application, when a device is having problems, it is difficult to identify the device because there is no data on where that device is located within the system. This makes addressing the problem difficult.

Currently, the AlliedViewTM NMS must be installed and configured before use and cannot be installed on mobile devices. The AlliedViewTM NMS is a Java Application which is meant to run on a Windows machine. Now that devices such as tablets and phones are common, it is necessary to accommodate these devices as well. Technicians in the field do not have a convenient way to view the status of each device geographically. Large businesses and enterprises face similar issues with navigating through their own equipment. Existing solutions do not capture a single building's network systems in a location based representation or map, which can be useful as networks get more complex.

## 3. Project Goals & Benefits

The major goal of the project is to create a web application that will allow users to view network equipment configurations in detail and interact with it. As mentioned above, the current system has to be installed and configured in order to use, and is not a web based solution. Our web application provides an easy to use and an interactive graphical interface that allows users to view necessary information about network devices, which is otherwise displayed in a tabular format. The graphical interface displays the exact location of devices on a floor in a building. The application allows users to place or remove icons of network devices on a blueprint of a floor and monitor the status of these devices using any modern web browser. The application does not need to be installed and is also accessible on mobile devices such as tablets and phones. It is scalable, modular and user friendly. The web application provides an interface that maintains the location of devices within a building, which can be expanded upon in the future.

## 4. Development Methodology

Our team used an iterative process for this project, having three iterations to complete the main requirements of the project. Each iteration was composed of a review of requirements given by our sponsors, design, implementation and testing. The duration of each iteration was approximately three weeks. In each iteration, we implemented both acceptance and unit tests immediately after implementation. At the end of each iteration, we showed a demo of the application and our progress to our sponsors. We had bi-weekly meetings with our sponsors to ensure that everyone were on the same page throughout the development process and to ensure that we are making proper use of time.

## 5. Challenges

**Technology**

Initially our team had to learn to implement front end technologies that were new to us, in order to develop the web application. We were able to overcome this by following many tutorials. Later in the project, we were having trouble saving blueprints to the back-end and saving layouts as a PDF.

**Testing Framework**

Due to the nature of some libraries such as Leaflet, it is difficult to perform unit testing on the front end effectively. We decided on using the QUnit testing framework initially, but the Leaflet library was not cooperating with QUnit. In order to overcome this issue, we depend on extensive black box and acceptance tests.

**Legal Issues**

We made sure that any external libraries or technologies we use in the development of the system complies with the terms of use set by the respective organizations.

# **Resources Needed**

*Author(s): Cameron Lanier*

**Software Needed for Web Application**

HTML 5 – To structure our web application.

CSS 3 – To style our web application.

JavaScript – To add functionality to our web application.

JQuery 1.12.0 – To add more functionality, include event handling and enable AJAX calls to the Rest API in order to send and receive data to/from the back-end.

Bootstrap 3.3.6 – Library to help develop and style responsive applications.

Leaflet 0.7.7 – JavaScript library to help create mobile-friendly interactive maps.

Spring Tool Suite – IDE built on Eclipse which provides different tools for development.

JUnit – Testing framework used to test our Java backend.

WebStorm – JavaScript IDE used for our development.

**Software Needed for REST API**

Java JDK 1.8 or later - To compile and run the Java REST application.

Spring Boot 1.3.2 – To configure the REST API including dependencies.

Apache Tomcat – Webserver to run the Java REST application on.

Developmentmachines have been obtained for development and testing of the application. A RHEL server is used for hosting the entire application.

# **System Requirements**

*Author(s): Cameron Lanier, Harsh Singh, Balaji Sundaram, Qiufeng Yu*

The Building/Location Equipment Map allows users to see all the network equipment in a building in a graphical manner. Using the application, users can easily view each floor and the status of networking equipment located in their respective locations. Users can upload and switch between blueprints of the different floors of buildings. They can place or remove device icons on different floors, creating layouts. These layouts can be printed, retaining the position of the devices. Different modes of operation ensure that the user has the ability to either edit or view changes.

## Functional Requirements

**Terminology**:

A blueprint is the image file of the actual floor plan of a building.

A layout includes the network devices placed on the blueprint.

A building is composed of floors that each have an associated layout.

Device “fields” refer to:

Name, IP Address, Subnet, Device Type, Version, Serial Number, Status, Release.

**1. Blueprint File Management**

**1.1** The system allows a user to upload a blueprint of a building for editing.

**1.2** The system allows the user to view a list of uploaded blueprints.

**1.3** The system allows the user open a blueprint.

**2. Creating a Network Map**

**2.1** The system allows the user to create and name a building.

**2.1.1** The system allows a user to select the number of floors.

**2.1.2** The system allows a user to assign a blueprint to each floor.

**2.2** The system allows the user to place devices on a blueprint.

**2.3** The system allows the user to move devices on a blueprint.

**2.4** The system allows the user to add/edit notes about devices.

**2.5** The system loads the fields of each device from the device list file.

**3. Viewing the Network Map**

**3.1** The system allows a user to open a layout.

**3.2** The system allows a user to view the status of devices on the map.

**3.3** The system allows a user to view all the fields of a device

## Non Functional Requirements

**1. System Response**

**1.1** The backend queries the device data file every 30 seconds.

**1.2** The web application is robust enough to handle 30 network devices on a single layout without a noticeable lag.

**2. Representation**

**2.1** The web application makes use of simple icons.

**2.2** Device icons avoid photo realism and are simplistic.

## Constraints

1. The REST API is created using Java 8.
2. The backend runs on Apache Tomcat 7.

# **System Design**

*Author(s): Cameron Lanier, Harsh Singh, Balaji Sundaram, Qiufeng Yu*

## High-Level Design



Figure Overall Design

Our design consists of two major parts – the front end client and the Java backend. Fig. 1 depicts a design of all the components in the system. The web application follows the Model View Controller design pattern, and sends information using HTTP requests to the REST API in the backend. The backend reads device status and other information from a text file named devicesInfo.txt. This information is then parsed and prepared to be sent to the front-end of the web application. When in view mode, information is automatically updated every 30 seconds using AJAX calls.

The View of the application is developed using HTML, CSS, and Leaftlet.js. Leaflet is a powerful open-source mapping library for JavaScript, which allows panning and zooming around maps and provides events for markers or, in our case, device icons. HTML is used for the page structure, and CSS is used for the design. The Controller implements both JQuery and Leaflet, which allows event handling, and listen for interactions with the map on all elements within the application such as buttons and device icons. The Leaflet events handling interactions make up the Controller. The Model is made up of data structures containing layout and blueprint information that are populated using JQuery. JQuery is also used to send AJAX calls to the back-end REST API to send and receive both layout and blueprint information as well as updated device information. This information is sent as JSON objects or arrays of JSON objects, with the exception of blueprint images, which are transformed into base64 representations and stored in the backend.

## Backend Design

The backend server utilizes the Spring Framework and runs a REST API that interacts with the front-end client. Conventional HTTP request methods are utilized to send and request data to/from the backend. The endpoints of our API are addressed later in the document, under the low level design section. The concentration is on the images of a building (blueprints) and the entire map with the devices, their location and status (layouts). The backend retrieves data from the deviceInfo.txt file whenever the call is made from the frontend, parses the data in order to keep the format consistent across the system, and sends it back. The backend serializes the JSON objects to Java objects using the Jackson library, which allows JSON objects to be easily manipulated as Java objects. Information about blueprints and the layouts are saved as files in directories specified by the program, with an easily identifiable name. The JSON objects allow for extensibility if we want to migrate to a database, since they can be parsed and stored in a database if needed. The list of available blueprints and layouts is stored in a separate file to keep track of the different files.

The backend also comprises of a REST API that is created in Java using the Spring Framework. Fig. 2 below shows a UML diagram of the backend system. It consists of classes which correspond to devices, layouts, blueprints, and buildings. The Building object has an ArrayList of Floor objects. The Floor objects have an ArrayList of Devices which stores device information. Each Floor object has a layout assigned to it which is saved in a text file. The Notes object corresponds to the notes that can be added to devices on the map. The Blueprint object contains the blueprint’s name, height, width, and the base64 string representation of the blueprint image.

The backend also consists of Controller classes, which are where the different requests are defined. BlueprintController, NotesController, LayoutController and BuildingController are the controller classes for the Blueprint, Notes, Layouts and Buildings classes respectively. The Application class is responsible for bootstrapping and launching the Spring Application from the main method. This class also creates all the required files and directories, if they haven’t been created yet.

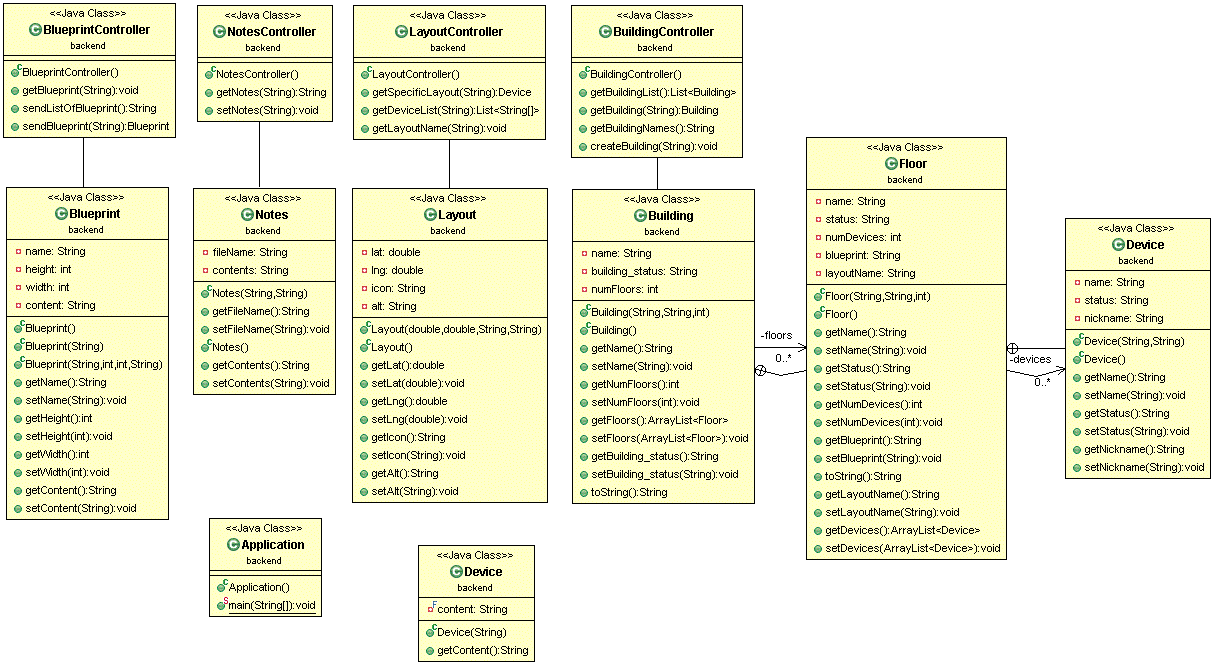
****

Figure 2 UML Design of Backend

## Low Level Design

**GUI**

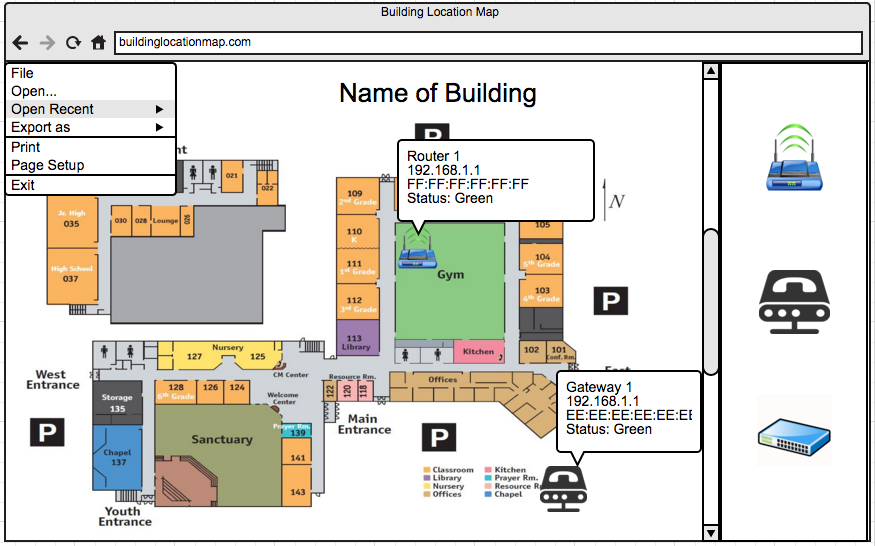


Figure 3.1 Initial GUI Mock Up

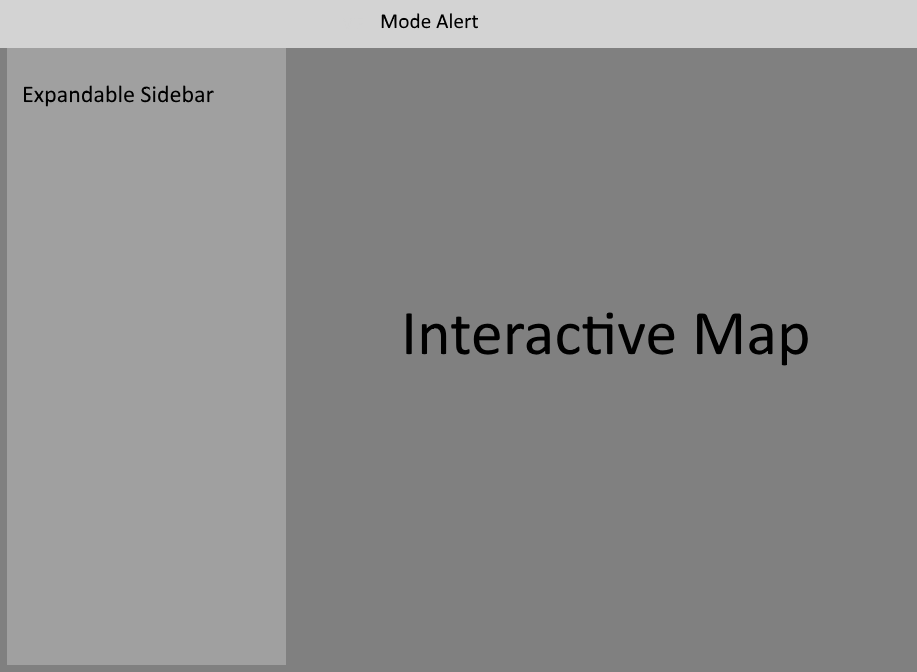
****

Figure 3.2 Current GUI Mock Up

The GUI consists of different sections: the Expandable Sidebar, the Mode Alert, and the Interactive Map itself.

The E*xpandable Sidebar* is composed of buttons for different views: the Building Menu, the View Layout, Edit Layout, and Layout Settings.

*Building Menu* provides the ability to create a building by giving it a name and specifying the number of floors. This menu option also allows a user to upload a blueprint image and give it a name. This image can then be applied to one or more floors of the building.

*View Layout* gives the ability to view all the buildings saved in the system. The buildings display their status beside their name. When a building is expanded, the different floor names and their statuses can be seen and when each floor is expanded, the devices on each floor and their statuses can be seen. These statuses are calculated and updated at a 30 second time interval. A building can have multiple floors, and each floor can have multiple devices.

The status of the building is calculated to be online only if all the devices on all floors are online. Even if one device is offline, the whole floor and the whole building show as offline. Different floors can be viewed from this menu, and the devices on the floor can be zoomed into from this menu as well.

*Edit Layout* provides the ability to select a building and load a floor from the building. Once a floor is loaded, devices from a list of all monitored devices can be added to the layout. The devices can be moved to their real world location, and then the layout can be saved. The added device becomes unavailable for adding to any other floor or any other building.

*Layout Settings* gives the user the ability to print the current map.

The M*ode Alert*section allows users to quickly see what mode they are in, in addition to seeing what building and floor they have currently loaded.

The *Interactive Map* section allows users to pan and zoom throughout the map, click on the different icons and move device icons when in edit mode. Information is displayed to the user by clicking on or hovering over the device icons on the map.

**Web Application**

Utilizes arrays and objects for sending and receiving data from the backend through the REST API. Details about the implementation is documented in the Implementation section.

**Backend**

Rest API Endpoints:

|  |  |  |  |
| --- | --- | --- | --- |
| **Endpoint** | **HTTP Verb** | **Post Body** | **Result** |
| */blueprinttable* | GET | none | Returns a list of all of the saved blueprints as a JSON string. |
| */blueprint/{name}* | GET | none | Returns a JSON string containing the details of the specified blueprint. |
| */blueprint* | POST | JSON String | Accepts a JSON string containing the details of a blueprint and saves it in the corresponding blueprint file. |
| */buildingList* | GET | none | Returns a JSON string representation of all buildings, their floors, the devices on each floor and their statuses. |
| */building/{name}* | GET | none | Returns a JSON string containing the details of the specified building. |
| */building* | POST | JSON String | Accepts a JSON string containing the details of a building and stores it in the corresponding building file. |
| */buildingListName* | GET | none | Returns a list of names of all the buildings. |
| */layout/{name}* | GET | none | Returns all the markers and their locations on the specified layout. |
| */setNotes* | POST | JSON String | Accepts a JSON string containing the notes for a device and saves it in the corresponding file. |
| */getNotes/{name}* | GET | none | Returns the notes for the specified device, if it exists. |
| */error* | *GET* | *none* | Displays an error page. |

Figure 4 Rest API Endpoints

**File Formats & Structure**

Buildings, layouts, blueprints, notes, and devices are all saved in separate directories which are generated if they do not yet exist. The figure below demonstrates the file hierarchy.

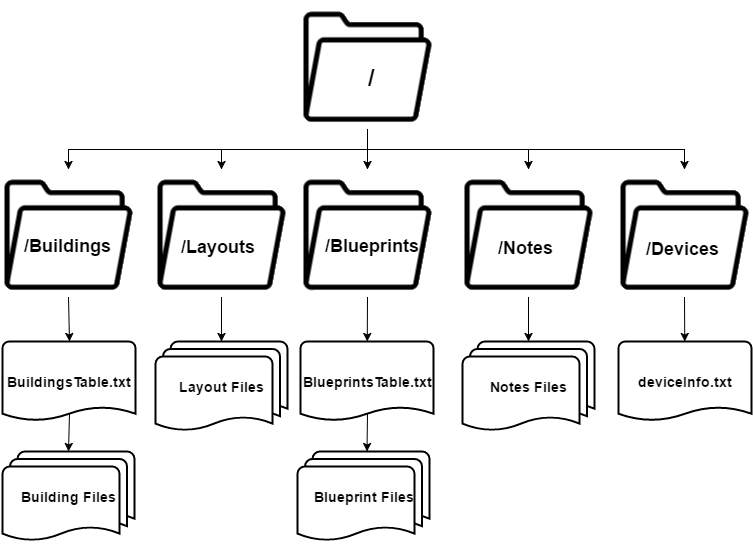


Figure 5 File Structure

Buildings

BuildingsTable.txt stores the names of the buildings that have been saved so far and is separated by a new line characters (\n). The figure below shows the representation of how data is stored in the file.



Figure 5.1 BuildingsTable Format

Each building file is saved in the format <buildingname>.txt, using the name of the building. They are flat files containing string representations of a JSON object representing a building. The figure below shows the structure of a building object.

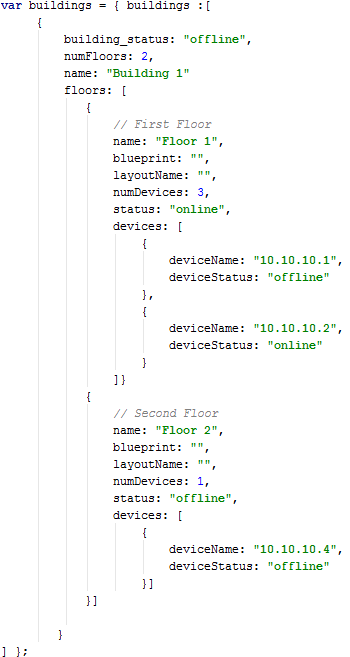


Figure 5.2 Building Object Structure

Layouts

Each layout file is saved in the format <layoutname>.txt, using the name of the layout that is formed by appending both the building name and the floor name. This file represents the layout containing all the devices on the respective floor. The data is stored as an array of JSON objects. The figure below shows the structure of a layout object.

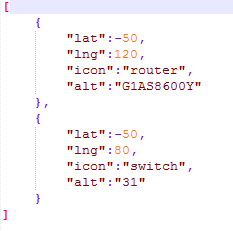


Figure 5.3 Layout Object Structure

Blueprints

BlueprintsTable.txt is also a flat file like the BuildingsTable.txt file, in which each saved blueprint name is stored separated by a new line characters (\n). The figure below shows the format of the file.



Figure 5.4 BlueprintsTable Format

Each blueprint file is saved in the format <blueprintname>.txt, using the name of the blueprint. It is a text file containing the name, height, width and the base 64 representation of the blueprint separated by commas.



Figure 5.5 Blueprint Format

Notes

Each notes file is saved in the format <deviceserialnumber>.txt, using the serial number of the device. These are text files containing the notes for each device when they are added. The notes are JSON objects, stored in a text file. The figure below shows the format of the file.



Figure 5.6 Notes Format

Devices

The deviceInfo.txt file holds the information about all the devices that are monitored by the system. Information about each device is separated by a new line characters (\n). Each information field is separated by a comma. The different fields are: device type, IP address, netmask, type, version, serial number, status, release, location in NMS, and device nickname. The figure below shows the format of this file.

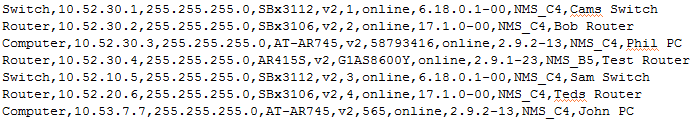


Figure 5.7 DeviceInfo Format

# **Implementation**

*Author(s): Cameron Lanier, Harsh Singh, Balaji Sundaram, Qiufeng Yu*

**First Iteration**

For our first iteration, we started off by doing some research on how we would approach the overall problem presented to us by our sponsors. We broke the overall requirements into multiple tasks and started to explore the different tools and technologies we would need to use.

Due to the extensive graphical nature of this project, our first plan of action was to draft up a mock GUI representation of our initial ideas. Figure 3.1 was our initial GUI representation of what we had envisioned.

This representation illustrates the basic functionalities that our product needed to entail. These rudimentary functionalities include the ability to:

* Upload and open blueprints.
* Place devices on a blueprint.
* View the status of each device.

All these features needed to be built upon an interactive map with marker/icon placement. We were able to find a JavaScript library named Leaflet.js, which had the mapping functionalities similar to Google maps but was flexible enough to allow us to use our own images for mapping. Upon discovering this library, we were able to begin developing a map interface.

After getting Leaflet up and running, we were quickly able to place markers on the map and save them. We also figured out how to reopen the map with the marker locations saved. Accomplishing these basic initial tasks helped us get more comfortable with the library and helped lay the ground work for the next iteration.

**Second Iteration**

The second iteration was composed of linking a sample device file to the markers and display data from them, separating the edit and view modes of interacting with the map, and changing and assigning floors to buildings. We were able to read from a file saved on the disk by issuing GET requests using AJAX. We were able to implement two separate modes for editing devices and viewing the devices on the map. We also started developing a basic REST API using Spring Tools so that we could load data from the file using an endpoint instead of having to open the file directly from the front-end.

**Third Iteration**

Our third iteration includes displaying blueprints and layouts on the web application and being able to switch between different layouts. In this implementation, the user can place routers, switches, and computers on the map. We have added the ability to switch between edit and view modes, where the edit mode allows the user to move devices around and the view mode restricts the user from editing the devices and only allows viewing them. The current implementation allows the user to switch between blueprints by clicking on buttons. The view mode allows the user to view the current list of layouts and provides the option to open previously saved layouts.

We were also able to successfully save and open layouts to/from the back end. The application makes POST requests whenever a layout is saved and the web application can get the list of current layouts saved in the system with a click of a button. The saving of the layouts is implemented under the edit layout section of the sidebar, a text box is provided for naming the layout and when the save button is clicked, the layout is saved. The user can load a new layout by typing in the name of a layout then clicking on the open button. The different layouts are stored within a text file as an array of JSON objects, which correlate to the different markers on the layout. When a new layout that has not been saved yet is sent to the backend after the name of layout is saved in a text file, a LayoutsTable.txt file is created, which stores the name of the layouts separated by a new line character.

**Fourth Iteration**

In our fourth and final iteration, we had to re-structure our classes in order to implement a hierarchy that we designed. We designed the Building hierarchy, in which each building has a floor which has a layout assigned to it. Each floor can have a number of devices on them. The Buildings can have a varying number of floors which can be selected by the user, and each floor is associated with a layout. We also set up polling of device information from the backend. In the view mode, the web application sends a GET request to the REST API which in turns updates the device information. We changed the way devices were being uniquely identified by changing the mapping from IP address to serial number, which opened up the possibility of adding notes to each device that are persistent across all sessions. This iteration also included various user experience additions such as automatically saving the current floor if a user navigated to a different floor without saving the current floor, showing prompts when actions are completed. Devices utilize a “nickname” for more descriptive identification especially when placed on the map. The text for the nickname also has a white shadow which allows for better readability. We also implemented a print functionality that allows users to click on the print button and print a copy of their current view of the map.

**The Code Base**

The code base will be split into two different sections, the Web Client and the Backend Application.

*Backend Application*

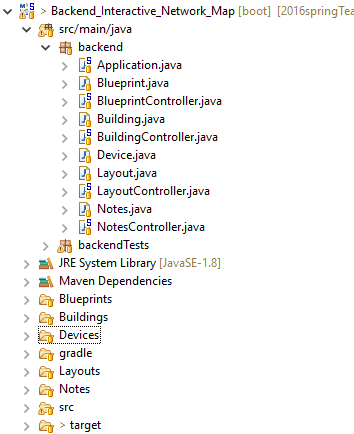


Figure 6 Backend File Structure

Utilizing the Spring Framework, we were easily able to get a REST API running within our Java Application. Apart from the framework, we also used Spring Boot to quickly create an application we were able to customize throughout development.

As mentioned above, Application.java holds the main method where the entire application gets launched as a Spring Application. It initializes Apache Tomcat and starts creating and mapping the different REST endpoints discussed in the Design section. Each Controller class is assigned as a REST controller by utilizing the @RestController tag at the top of the class. Each method in the controller includes both a @CrossOrigin tag that signifies that a request on this endpoint can come from any location and a @RequestMapping tag which is used to notify the controller that the method can be used as a REST end point.

The BuildingController class has a method called getBuildingList, which is responsible for calculating the status of each level of the Building hierarchy. This method gets the status of every device, on every floor, in every building and returns a list of Building objects. It calculates the status of the object which is above it in the hierarchy using the status of object below it. This is done by opening the file of every building, iterating through each floor and opening up the file associated with that floor. Each layout file is then serialized by using an ObjectMapper from the Jackson library, which enables JSON objects to be represented as Java Objects. These devices are then iterated through to find their current status from the deviceInfo.txt. The ObjectMapper requires that the names of variables in the Java classes match exactly with the JSON object in order to map objects successfully.

Implementing the interaction with different files was relatively straight forward. When a call is made to a method which needs to interact with a file not in JSON format, such as BlueprintsTable.txt (and all other \*Table.txt files), a standard file scanner is utilized. However, when a JSON formatted file is saved, the ObjectMapper utilizes the readValue method to map the objects to a class, as seen in Figure 7 below. This makes converting from JSON and Java objects quite easy to implement, and can later be utilized for saving data into a database if needed.

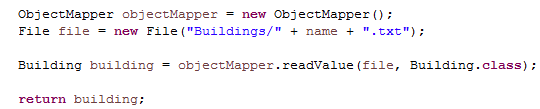


Figure 7 Jackson ObjectMapper Example

Files are named appropriately and consistently throughout the application and are written to using the PrintWriter class. This is the case for the non-JSON strings and files much like the BlueprintsTable.txt and others. The ObjectMapper class also has a method to write Java objects as JSON strings, which we have taken advantage of for many class objects.

*Web Client*

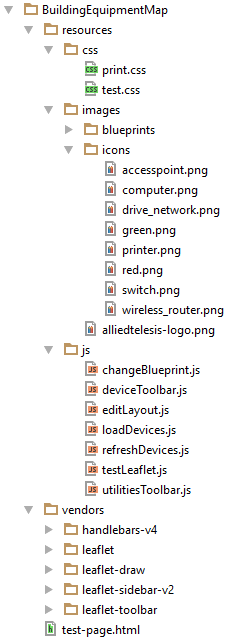


Figure 8 Web Application File Structure

The code making up the Web Client is split up by functionality. The resources folder contains folders for css, images, and a js folder that contains the main JavaScript code base. This folder names are pretty self-explanatory. The CSS file is located in the css folder. The images folder contains all the images of icons and the status images. The js folder contains the JavaScript code.

The testLeafltet.js class creates the Leaflet map and is responsible for setting up the map for allowing interactions. The changeBlueprint.js class contains code for changing the blueprint and layout when different floors are chosen through the application. It does this by making an AJAX call to the backend, and receives the blueprint to load the leaflet map associated with it. The editLayout.js class is responsible for loading all the buildings to display on the sidebar. It is also responsible for loading a layout when the corresponding floor button is clicked on the application. The loadDevices.js class makes a call to the backend to get all the device information, assigns each field to its corresponding variable, and loads it into an array of similar objects that is used by other functions and classes. The refreshDevices.js class uses the setInterval() function to make call the loadLayout() function and buildHierarchy() every 30 seconds. This refreshes the current layout, if in view mode, and reconstructs the building view list shown on the sidebar. The figure below shows this code.

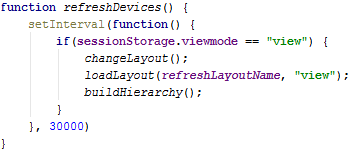


Figure 9 refreshDevices() function

The utilitiesToolbar.js class is responsible for the printing of the current view of the map. It strips down some of the sidebar elements to ensure that the user is able to print relevant content from the map. The deviceToolbar.js class contains the most code and is responsible for the majority of actions that take place within the sidebar of the web application. As mentioned previously, all device information is stored in an array of device objects called deviceInfo. The items array holds all of the current layout’s devices. The markers array holds all the actual markers and is different from the items array as it holds the actual leaflet marker objects, instead of just the coordinates and the serial number. When a new layout is selected, the objects in the items array are used to create markers, which are assigned different Leaflet events and are pushed to the Leaflet map as layers. The different icons of devices are created here and are defined as HTML strings within Leaflet. The switching between edit and view modes are also defined within this class. Upon switching modes, the map and the markers are redrawn with the markers’ “draggable” option set to true when in edit mode. Device icons are linked to the device information in the backend based on the serial number of devices. AJAX calls are also utilized to send the created building objects to save them to the backend, similar to the sending and saving of blueprints. When devices are added to the map in the edit mode, they are also removed from the drop down so that they cannot be added to another layout or another building, since the devices are unique. When a user clicks on a device, a modal displaying all of the information about the clicked device is displayed, and an AJAX call is made to retrieve notes for the device. The Handlebars templating engine for creating dynamic html templates is also defined in this file. The building object as seen in Figure 5.2, is used for creating the HTML for the building view on the sidebar.

The vendor folder contains some of the libraries which we used in the development of this project, including Handlebars, Leaflet, Leaflet-sidebar, Leaflet-draw, and Leaflet-toolbar. All of these libraries are free to use and have acceptable licenses for use in this project.

The test-page.html file is our HTML file for the entire web application. All the external libraries and the CSS file are included and linked in the head section of the HTML file. The sidebar is implemented using the sidebar div id, which is specified in the leaflet-sidebar-v2 docs. The HTML file also implements the Handlebars template that was defined in the deviceToolbar.js file, by iterating through the items using Handlebars helpers. An example of this is the “{{#each floors}}” command, which can be seen in the figure below.



Figure 10 Implementation of the Custom Handlebars Function

# **Testing**

*Author(s): Cameron Lanier, Harsh Singh, Balaji Sundaram, Qiufeng Yu*

Both acceptance and unit tests have been implemented to test the robustness of our application. After each iteration, we performed both acceptance and unit tests before showing a demo of the application to our sponsors. We used JUnit to perform unit testing on our Java code.

Test data is provided as text files in corresponding folders. This will ensure that the various classes can be properly tested using the appropriate data.

**Acceptance Test Plan**

Unless otherwise stated, all tests have the precondition that the web application is running.

|  |  |  |  |
| --- | --- | --- | --- |
| **Test ID** | **Description** | **Expected Results** | **Actual Results** |
| 1. testCreateNumberOfFloors | Precondition:  Make sure you have at least one PNG image ready to upload as the blueprint for one of your floors. Make sure this blueprint is already uploaded.  Steps:   1. Click on the building icon on the sidebar. 2. Type in “Building1” as the Building Name in the Building Name Field 3. Type “1” for number of floors 4. Select the blueprint layout you previously uploaded. 5. Click the “Save Building Button” | There should be an alert that comes up and notifies that the building has been successfully created. |  |
| 1. testCreateBuilding | Precondition:  Make sure you have at least one PNG image ready to upload as the blueprint for one of your floors.  Steps:   1. Click on the building icon on the sidebar. 2. Type in “Building1” as the Building Name in the Building Name Field 3. Type “2” for number of floors | There should now be two drop down bars available named “Floor 1” and “Floor 2”. |  |
| 1. testAddDevice | Precondition:  In edit mode. There’s a building selected and a floor loaded.  Steps:   1. Select the devices available in the drop down menu under “Add Devices” 2. Select “Router – Teds Router” 3. Click the button “Add to map” | The Router is now displayed on the map with a green status marker. This specific router should now also be removed from the drop down menu bar. | The Router is now displayed on the map with a green status marker. This specific router should now also be removed from the drop down menu bar. |
| 1. testViewDevice | Precondition:  deviceInfo.txt is loaded on the backend. Teds Router is placed on the Map.  Steps:   1. Select the View Layout Button 2. Select the Building and Floor of which it was placed on the menu 3. Select the Device Icon on the map | The extended information view is presented.  The values are as follows:  **IP Address:**10.52.20.6 **Netmask:**255.255.255.0 **Type:**SBx3106 **Version:**v2  **Serial Number:**4  **Status:**online **Release:**17.1.0-00 **Location:**NMS\_C4 | Successfully displays the device information.  The device info displayed is:  **IP Address:**10.52.20.6 **Netmask:**255.255.255.0 **Type:**SBx3106 **Version:**v2  **Serial Number:**4  **Status:**online **Release:**17.1.0-00 **Location:**NMS\_C4 |
| 1. testSwitchLayout | Precondition:  testEditMode has been just run as a precondition. BuildingOne is created which has different floors corresponding to the different layouts included in the Blueprints Folder. Floor One assigned to blueprintGray, Floor 2 assigned to blueprintTest, Floor 3 assigned to officeLayout  Steps:   1. Click on pen icon to get into the “Edit Mode”. 2. Change floor layouts by clicking on Floor 1, then Floor 2, and Floor 3. | The view of the application changes and it switches to the respective floor assigned with the proper blueprint. Floor 1 to blueprintGray, Floor 2 to blueprintTest, and Floor 3 to officeLayout. | Clicking each button for each floor successfully displays the respective blueprint on the application. |
| 1. testSaveLayout | Precondition:  Building One is created with one floor assigned to the officeLayout blueprint.  Steps:   1. Click on pen icon to get into the “Edit Mode”. 2. Select Building One 3. Load Floor 1 4. Select Computer – John PC, and then Add to Map 5. Select Save Floor | The message Floor 1 Saved! appears  When the floor is loaded, Computer – John PC appears | The message Floor 1 Saved! appears  When the floor is loaded, Computer – John PC appears. |
| 1. testOpenLayout | Precondition:  testSaveLayout has just been run.  Steps:   1. Selects View Layout 2. Select Building One 3. Then load Floor 1 4. Check Results | The layout Building One is loaded to the user.  John PC is loaded on the map | The layout Building One is loaded to the user.  John PC is loaded on the map |
| 1. testEditMode | Precondition:  (none)  Steps:  1. Click on the pencil icon to enter “Edit Mode”. | An alert should appear which states the user is now in edit mode and the name of the building | An alert is shown atop of the main screen which states Now Editing: Building One – Floor 1. |
| 1. testViewMode | Precondition:  (none)  Steps:  1. Click on the eye icon to enter “View Mode”. | You should now get a pop up window that says “Now Viewing” | An alert is shown atop of the main screen which states Now Viewing: Building One – Floor 1. |

**Unit Test Suite**

We have been able to achieve more than 80% of code coverage on our backend using JUnit. Test results are displayed in Figure 11.

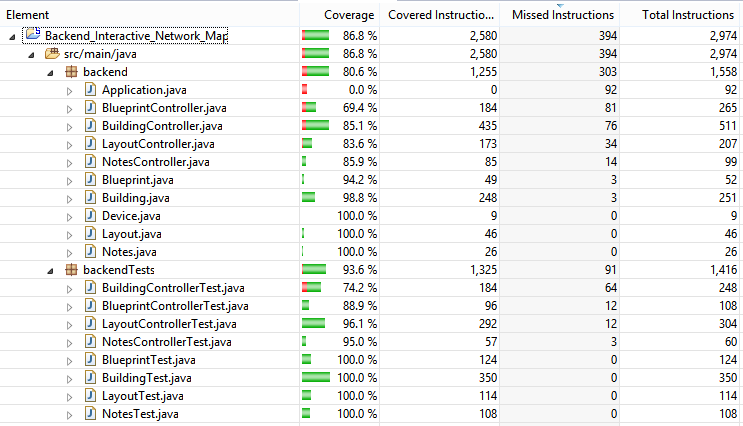


Figure 11 JUnit Test Results

# **Suggestions for Future Teams**

*Author: Balaji Sundaram*

Leaflet is a powerful JavaScript library with plenty of documentation online. Reading through the documentation and looking at examples would help get a grip on the library and its functionalities. The Leaflet documentation can be found here <http://leafletjs.com/reference.html>

The Spring Framework also has good documentation and many examples to refer to. Looking at these examples would help understand how RESTful services work. Our backend is built based on a simple REST service example documented in the Spring Tools Suite documentation. It implements many tools which helped us build this project

# **Developer’s Guide**

*Author: Cameron Lanier*

The ability to extend or modify the project later was kept in mind during development. We did not choose to use a specific framework for the frontend for this reason, since we are able to fully customize the majority of the code. For the backend, we chose the Spring Framework for helping us create the REST API.

## 1. Front End Web Application

The code is well documented, and should be relatively easy to read. JavaScript files are broken down by functionality.

Below is the Project Directory for the Web Application

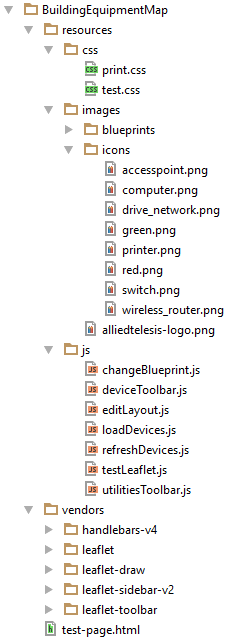


Figure 1 Web Application Framework

When adding new JavaScript files, remember to add the scripts to the test-page.html at the bottom to ensure they are linked to the application, since we do not have a framework responsible for this.

If new icons are preferred, they can be added in the icons folder, then the path name can be changed within the web application, to generate that type of icon. The actual icons should be at least 50px by 50px to ensure that there is not any pixilation, due to that being the basic size of the icons on the map. They can be added by emulating the function for createSwitchIcon and then altering the file path within the web application. This method is located in the deviceToolbar.js file. The addition needs to also occur throughout the code where icons are generated, such as in the methods loadLayout. The newly added icon should be added to the case statement. In the backend one must remember to ensure that the first parameter in the devicesInfo.txt is in fact the name of the type of icon or else the device will not be created with the correct icon.

Standard blueprints can be changed in the same manner, mainly for the first base image which initially is created.

The rest of the application is fairly straight-forward, with an understanding of how Leaflet, HTML, JavaScript, and Handlebars.

If one would like to point to the API, change the global variable which corresponds to the backend API. This global variable is located within the testLeaflet.js titled localhost found at the beginning of the file.

The JQuery API Documentation can be found here <https://api.jquery.com/>

The Leaflet Documentation can be found here <http://leafletjs.com/reference.html>

The Handlebars Documentation can be found here <http://handlebarsjs.com/>

## 2. Backend Java Application

This code is also well documented. The Java backend application was adapted from a Spring Boot REST API application. It utilizes Maven as its build tool, and utilizes the Spring Framework for the REST API.

If one would like to create other REST endpoints, mimic the creation of how they were created for those before them.

You can easily edit and add more devices to the devicesInfo.txt field by following the current naming conventions. There is currently not any proposed limit on how many devices that the system can hold and keep track of.

If you would like to add any new members to the default Table.txt files for testing, you can manually type the name in the file followed by a newline.

Layouts are saved in a folder named Layouts, while the blueprints are saved in a folder named Blueprints. Buildings are saved in a folder named Buildings, and notes for devices are saved in a folder called Notes. The devicesInfo.txt file must be present in the Devices folder in order for the system to populate the application with devices. The Java application finds and loads these files whenever they are requested by the web application.

The Spring Framework helps when attempting to modify the code for the RESTful API, and can be found here <https://spring.io/docs/reference>

One can import their project in from the zip file in eclipse, and run it that way, or they can utilize maven in order to create a jar file which can be run anywhere that has Apache Tomcat installed.

The command to run in maven to package the jar, first one has to install maven for this we used Apache Maven 3.3.9 and then navigated to the directory of the java application and ran “mvn package” which then creates a jar within the target folder.

To execute the jar type: java -jar “name of applicationr” by default the REST API will be on port 8080, this can be configured by typing in –server.port=<portnumber> on command line.

# 

# **User’s Guide**

**Introduction**

Users are able to create a building, and within the building create layouts of the floors and add networking devices on the different floors of the building. They would be able to save the layouts using the easy-to-use menu items and buttons on the side of the web application. Navigating the menu should be a simple task, with different menu items available for navigation. First and foremost, it is important to understand the three main levels of hierarchy: buildings, floors, and devices. These are discussed in detail in the sections below.

By interacting with the buttons as seen in both Figure 1 and Figure 2, users are able to place devices on the map and save the different layouts.

**Buildings**

Buildings are the top most level in the hierarchical order in the application. Buildings contain one or more floors. The application allows the user to create as many buildings as desired. By navigating to the Buildings sub menu, the user can create a building by filling out the name field and selecting a number of floors as shown by Figure 1.

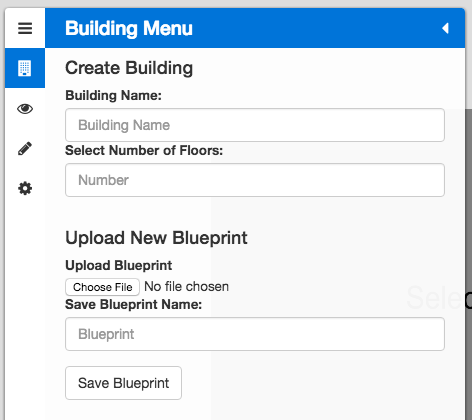


Figure 1 Building Menu

**Floors**

The next level of the hierarchical order is floors. There are one or more floors within a building. Floors, in turn, contain devices. Once a building is created, a user can upload a blueprint to represent a floor. The user has the option to assign one blueprint to multiple different floors or upload a different blueprint for each floor.

**Devices**

The last level of the hierarchical order is devices. Devices are the various types of network devices that the user can place onto the floor. The devices used in this application are represented by Figure 2. These are easily swappable for a different set of devices for the future use of the application. Devices also have extended device information illustrated by Figure 3.



Figure 2 Monitored Device

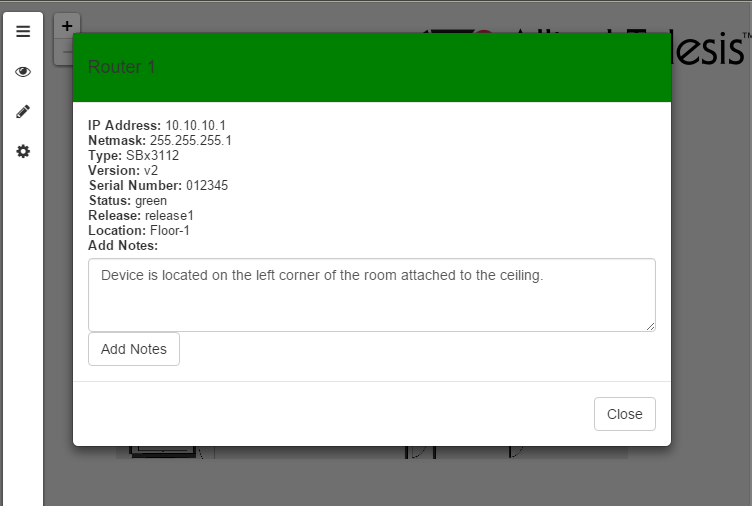


Figure 3 More Device Information

**Status**

Buildings, floors, and devices all have statuses – red or green – one indicating a problem and the other indicating no problem, respectively. The status of the building and floor is determined by the individual statuses of each device on the floor. If all devices on a floor are green, the floor will be green, and vice-versa. Furthermore, if all the floors in a building are green, the building will in turn be green as well. However, to change the floor status from green to red, at least one individual device has to have a red status. This also applies for the building, if one floor is red, the entire building will be red. Red is “offline” and green is “online”, and the markers are shown by Figure 4.

/Users/hsingh/Desktop/Screen Shot 2016-04-30 at 3.40.59 PM.png/Users/hsingh/Desktop/Screen Shot 2016-04-30 at 3.40.53 PM.png

Figure 4 Statuses

**Creating a Building**

After launching the app, the user will see the welcome screen which prompts the user to begin creating a building. The welcome screen is shown by Figure 5.

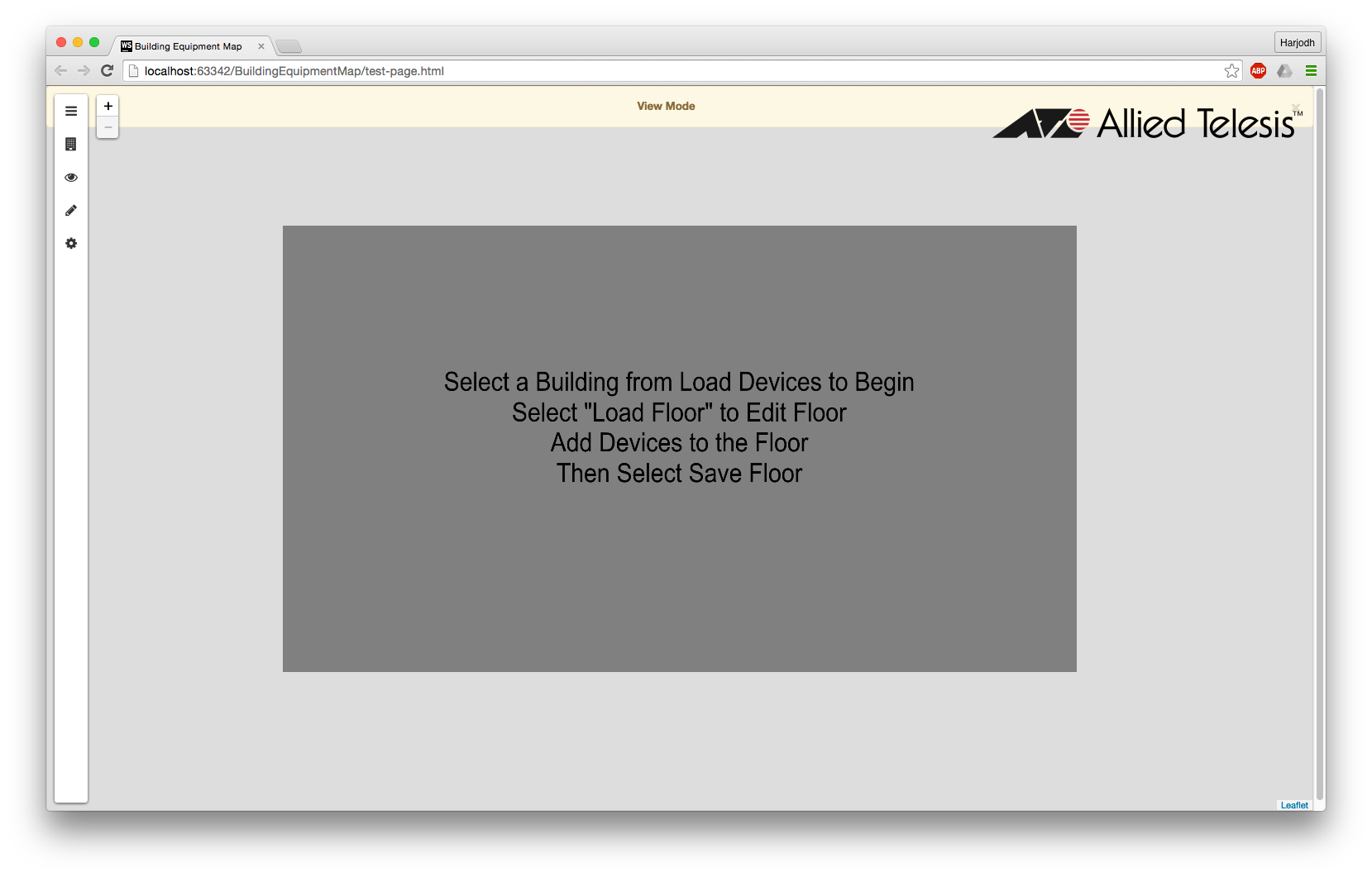
****

Figure 5 Welcome Screen

From the welcome screen, the user will click the building icon/Users/hsingh/Desktop/Screen Shot 2016-04-30 at 8.08.49 PM.png, which will open the Building Menu as shown by Figure 6.

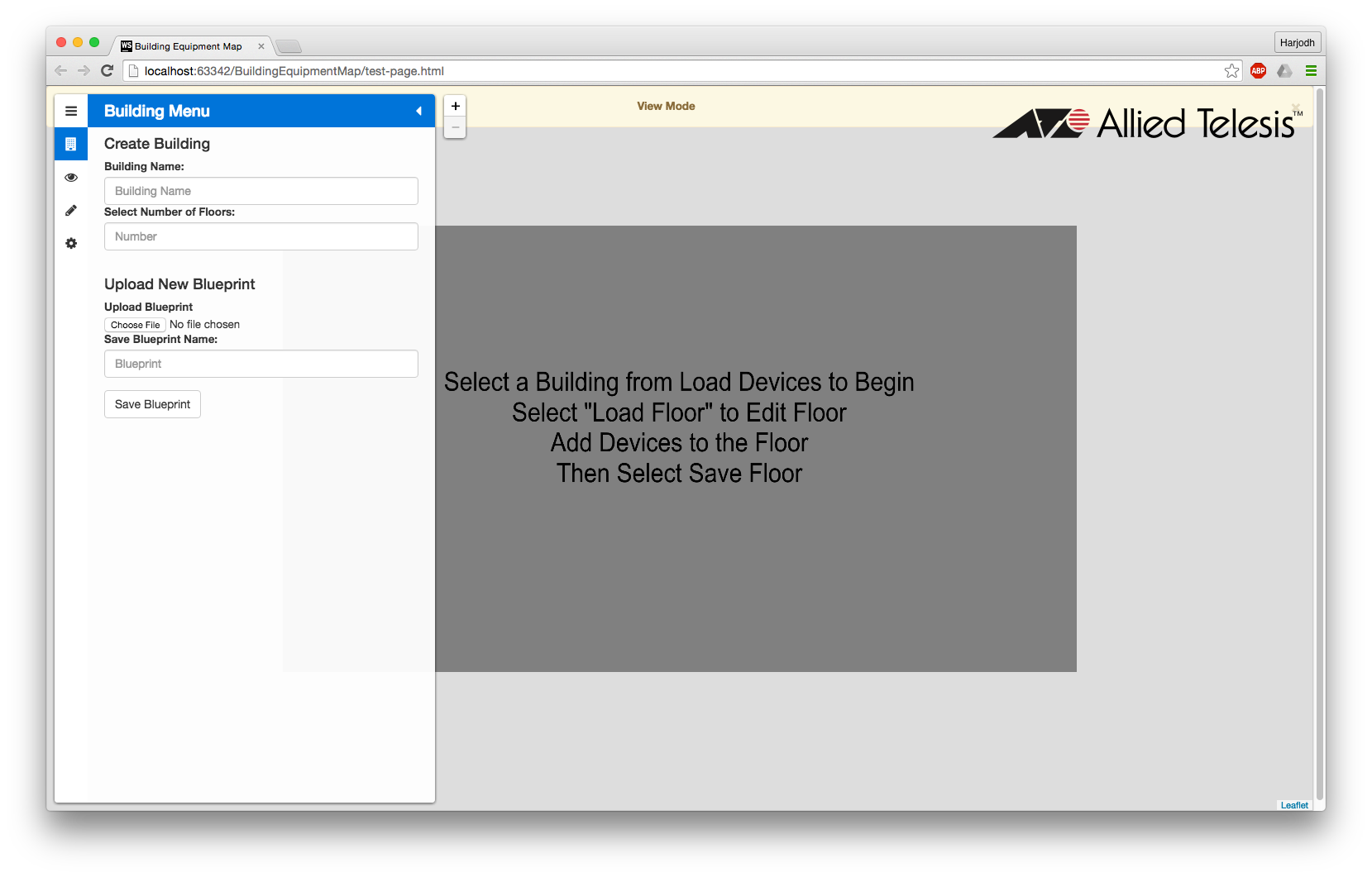
****

Figure 6 Building Menu

The user can now enter the Building Name and choose the number of floors. Next, the user can upload a new blueprint by clicking “Choose File”. The user can now name the uploaded blueprint and save it.

Next, the user can assign the uploaded blueprint to each floor. The user can use one blueprint for multiple floors or upload a new one for each different floor. Figure 7 illustrates this.

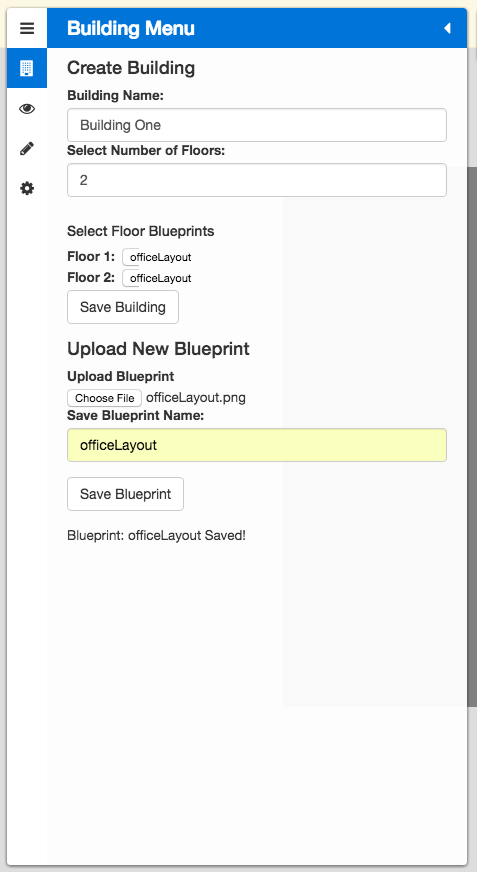
****

Figure 7 Creating a Building

As shown above, the same blueprint has been assigned to both floors. After this, the user simply clicks “Save Building” to create the Building.

**Editing a Building**

Now that a building has been successfully created, we can begin editing the floor blueprints. To begin editing, click on the pencil icon /Users/hsingh/Desktop/Screen Shot 2016-05-01 at 1.46.32 PM.png which will open the **Edit Layout** panel. From the drop down menu, select a building to edit. Now to begin editing a floor blueprint, click the Load Floor button next to the desired floor. Figure 8 below has Building One selected and Floor 1 loaded. This is also shown the by the yellow banner header at the top of the page.

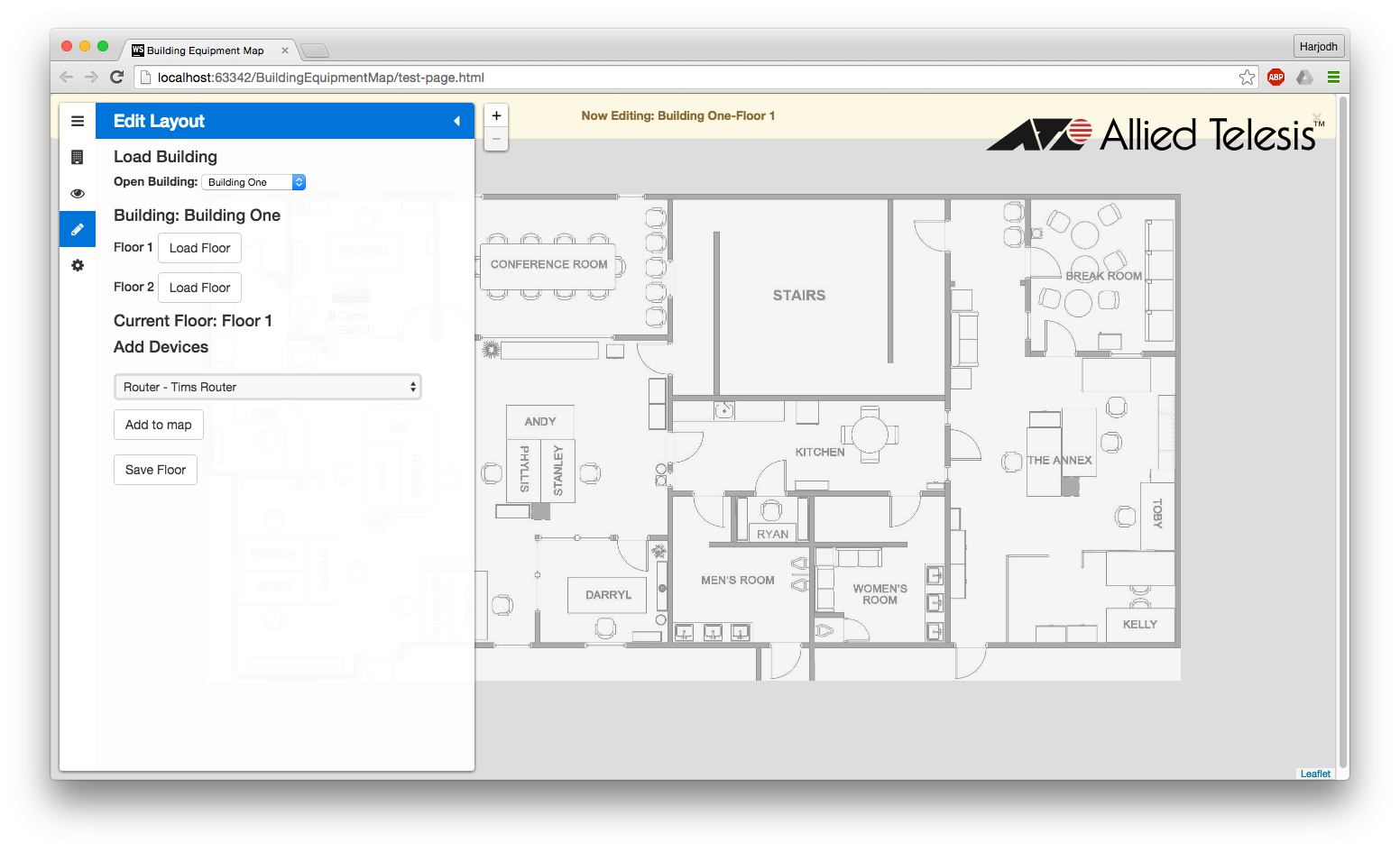
****

Figure 8 Editing a Floor

It is time to now add devices to the blueprint. Simply choose a device from the drop down menu under **Add Devices.** The number of devices in the drop down are currently loaded up from a file in the backend. The total number of devices avaisortinglable in this menu are currently contingent upon how many devices are actually added to the database (or file) in the backend. After selecting a device, simply click **Add to map** to place the device onto the map and then drag-and-drop to the desired location. Figure 9 shows the first device “Router - Tims Router” (in Figure 8) added and moved to the center of the map. When finished adding devices, the user can save the layout by clicking the button **Save Floor**.

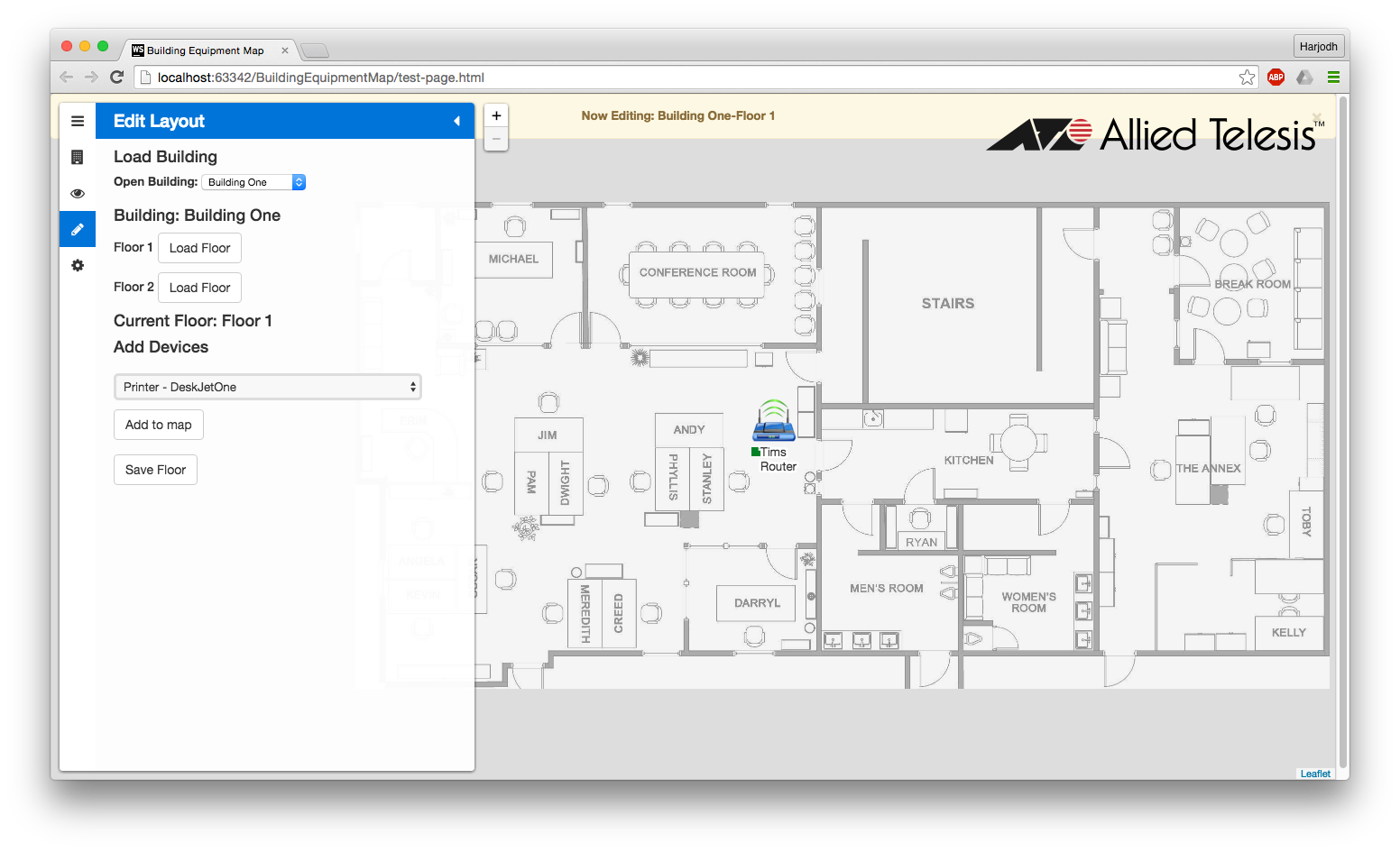
****

Figure 9 Adding a Device

**Viewing a Building**

After editing a building, it can now be viewed by clicking the on the eye /Users/hsingh/Desktop/Screen Shot 2016-05-01 at 2.48.37 PM.pngicon. This will open up the **View Layout** panel which will list all the buildings that have been created and their statuses. By clicking on a building, a drop-down panel will open displaying all the floors within the building along with the floor’s status. The Floor name can be clicked on to further expand the panel into the individual devices and their respective statuses. The **Open** button next to each floor can also be clicked on to open the respective floor layout.

When the floor subpanel is extended to display all the devices on that floor, each device can be clicked to locate it on the layout and it will pop up its status. These functionalities are illustrated in the figures below (Figures 10 & 11).

Figure 10 is expanded all the way (Building > Floor > Devices). This particular building has no devices that are offline (red), therefore the entire building is online (green).

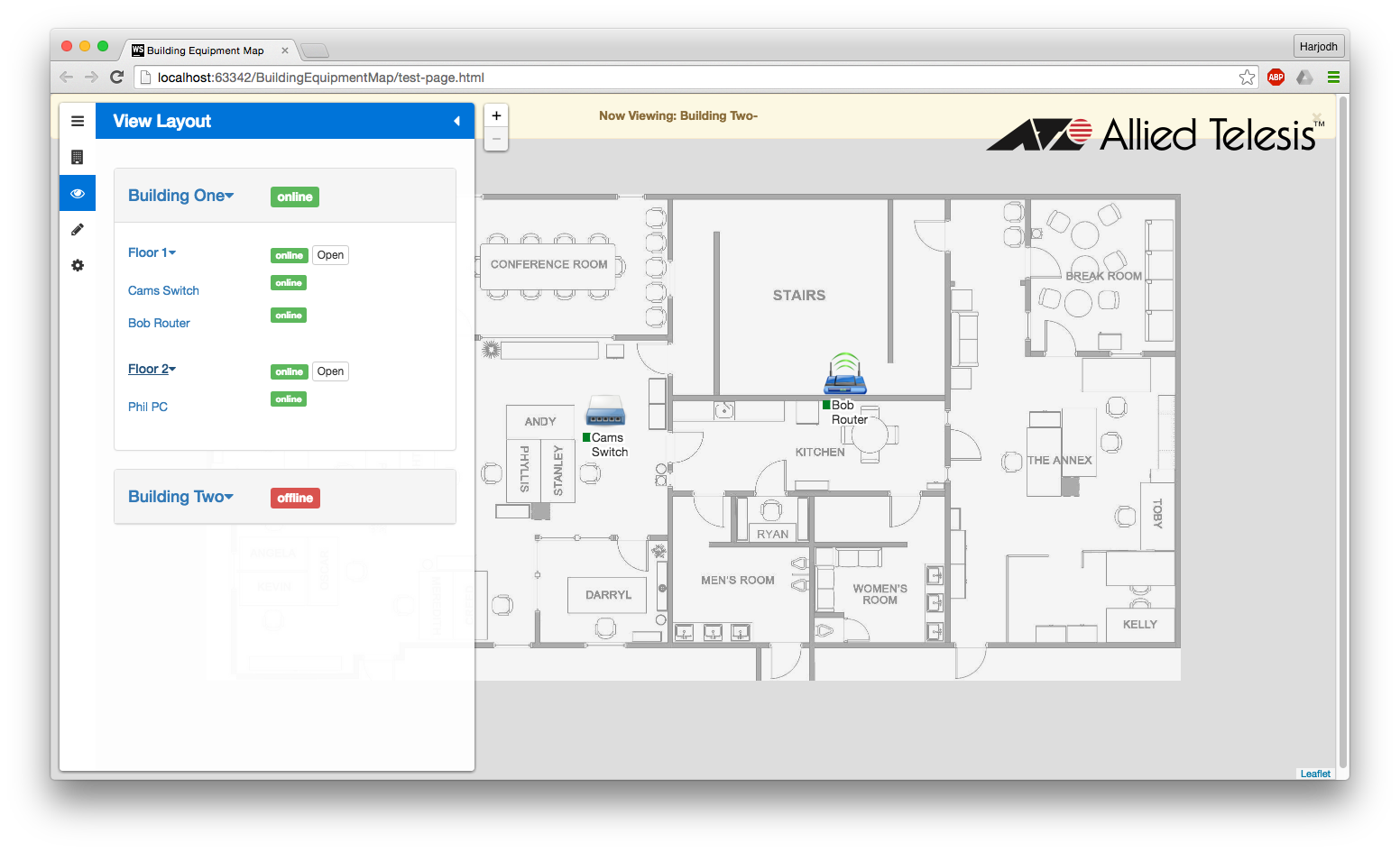


Figure 10 Viewing Building 1

Figure 11, however, does include one device on Floor 2 that is offline, making the entire building’s status offline.

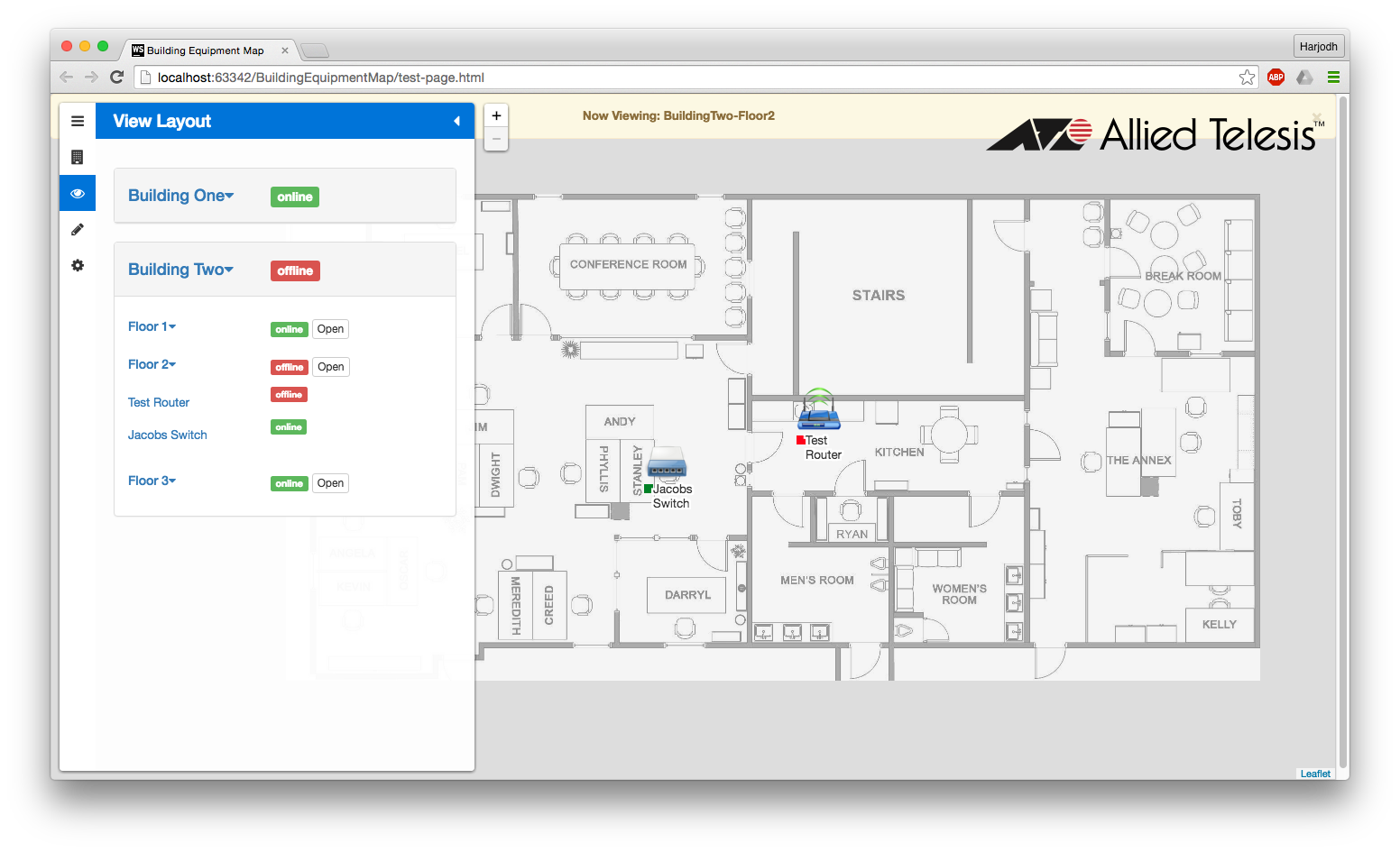
****

Figure 11 Viewing Building 2

By hovering over an individual device on the layout or clicking on it from the side panel, the device icon will show a small pop window with its **IP Address** and **Status**. This is shown in Figure 12.

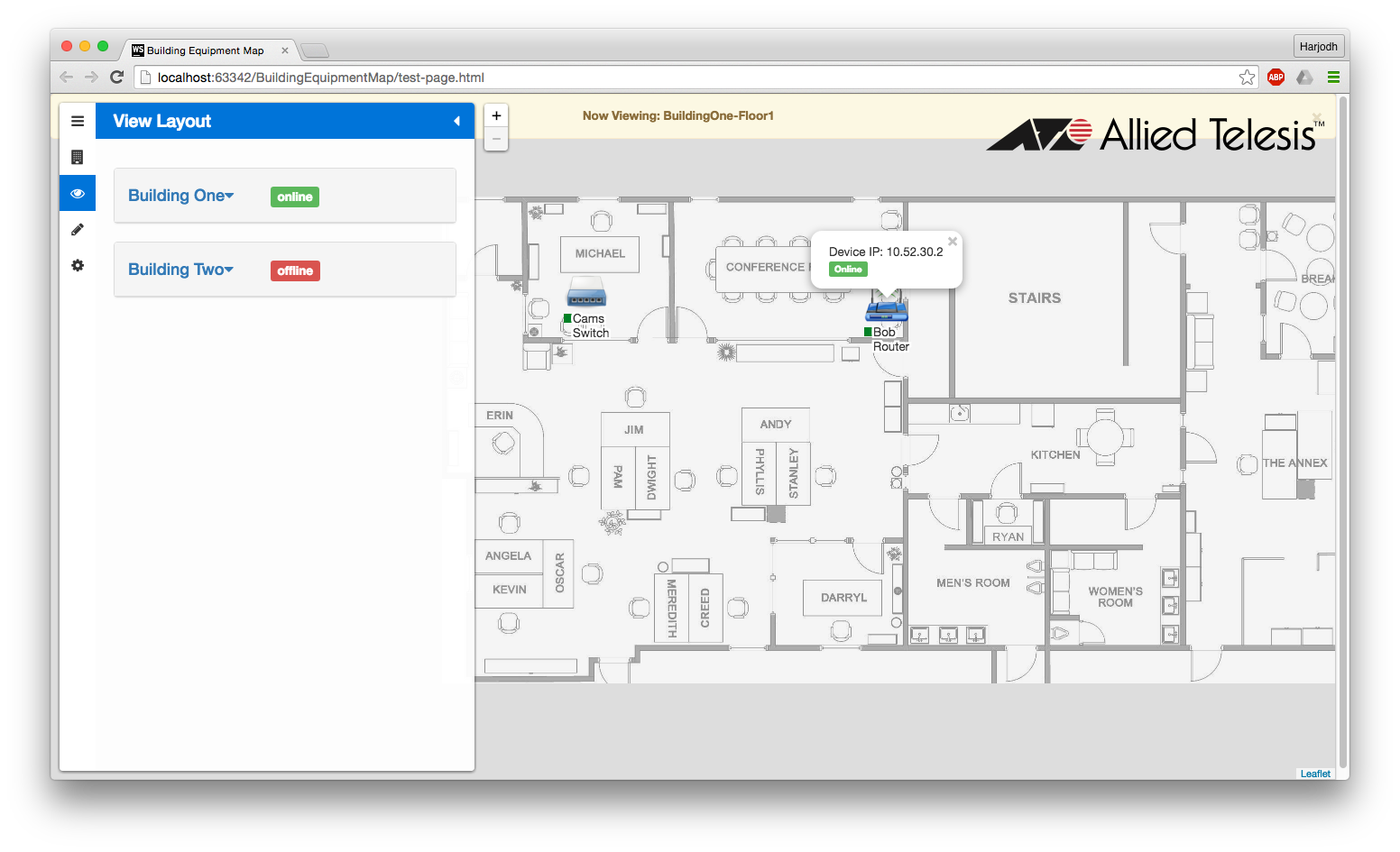
****

Figure 12 Individual Device Status Popup

**Printing a Layout**

In the settings menu, the user has an option to print a zoomed in version of a layout. The user can access the settings menu by clicking on the settings cog icon/Users/hsingh/Desktop/Screen Shot 2016-05-01 at 7.17.32 PM.png. The user will have to zoom in one level onto a device to activate the **Print Map** button. Figure 13 shows the **Layout Settings** menu and **Print Map** option.

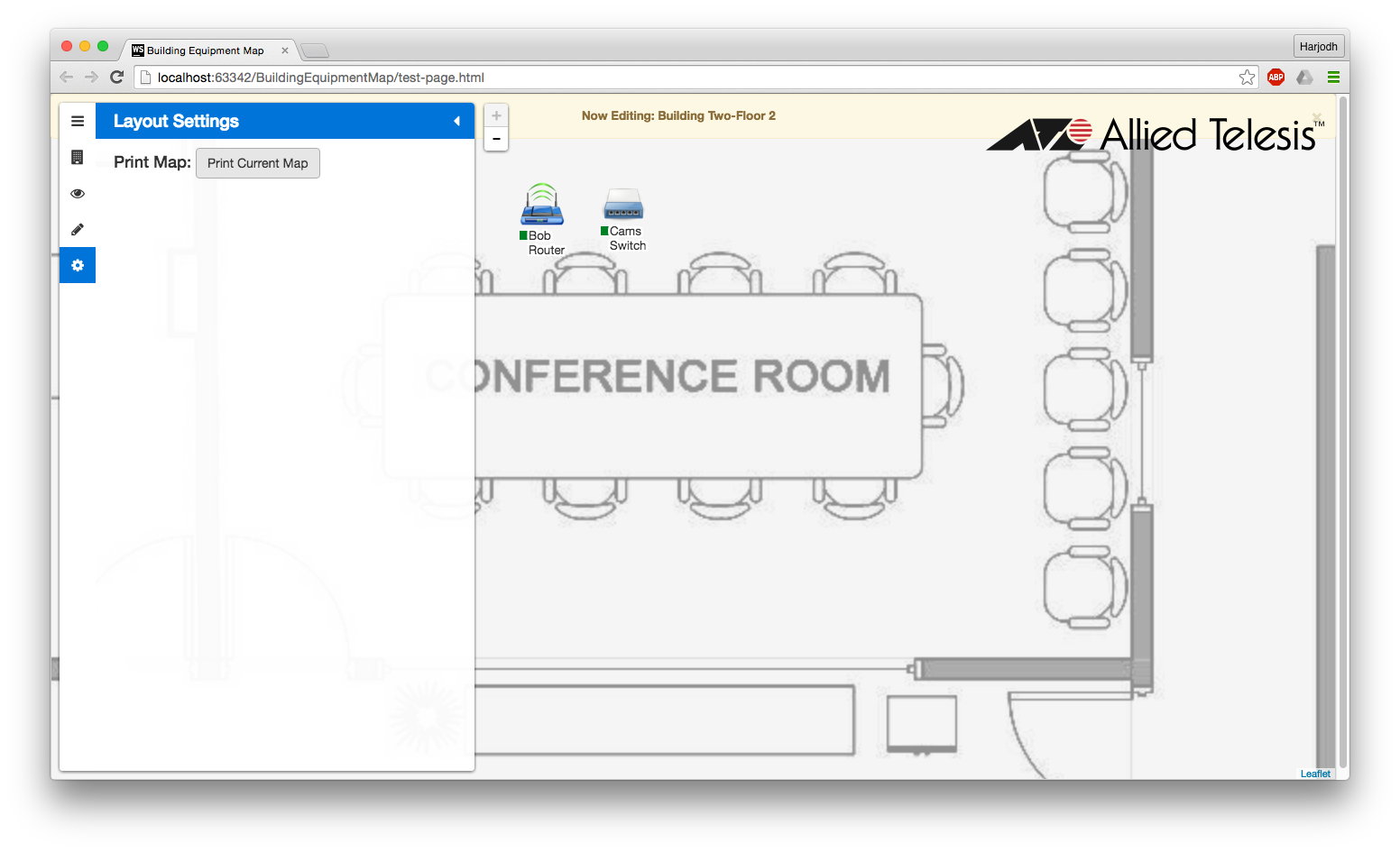
****

Figure 13 Layout Settings

Figure 14 shows the system print command window pop up when the **Print Current Map** button is clicked.

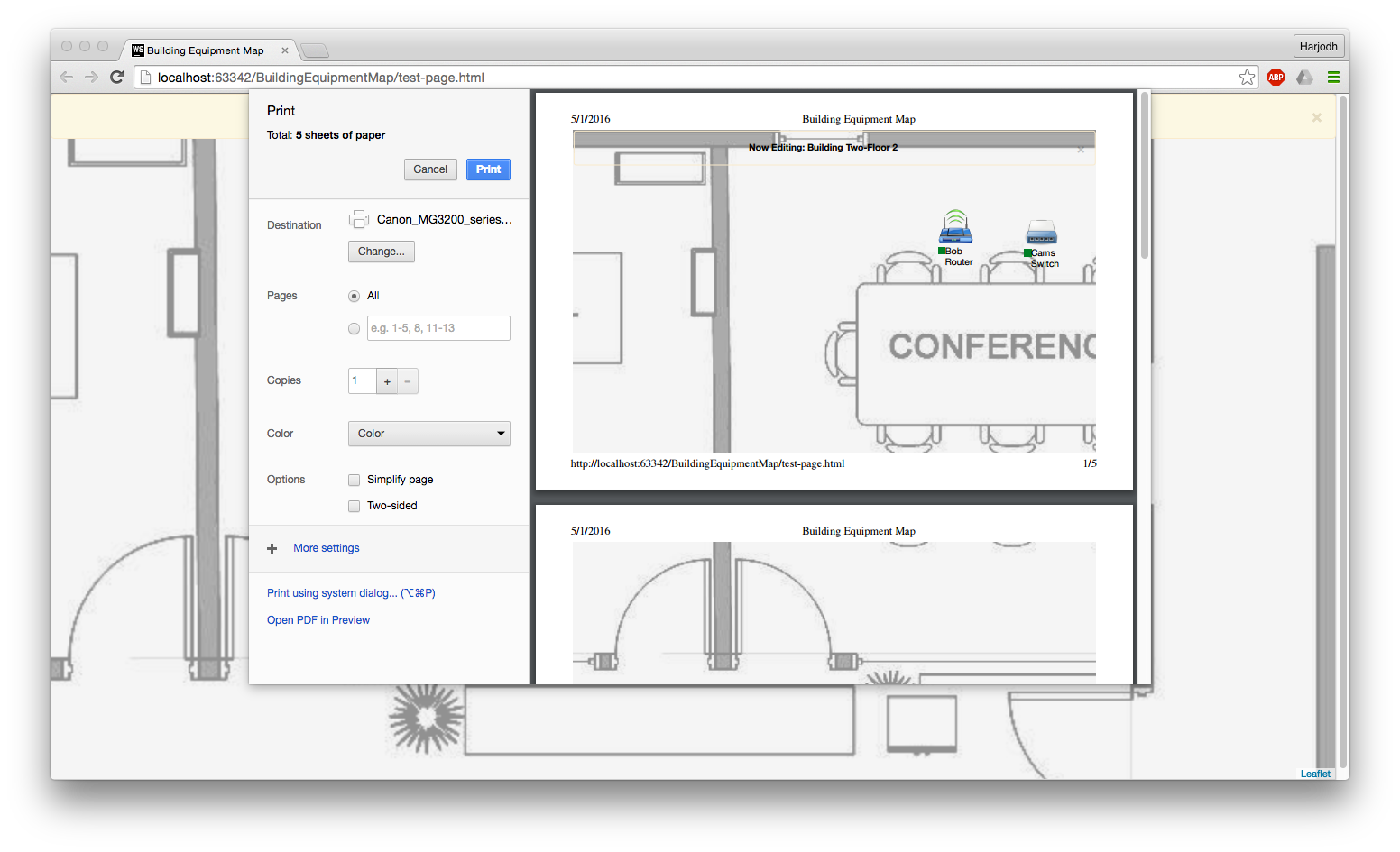


Figure 14 Print Layout Window

**Installation Guide**

**Required Software**

* Java 1.8 +
* Apache Maven 3.3 +
* Apache Tomcat 7 +

**Installation Instructions**

* Redhat Linux
  + Java 1.8: <http://tecadmin.net/install-java-8-on-centos-rhel-and-fedora/>
  + Apache Maven 3.3 +: <https://maven.apache.org/install.html>
  + Apache Tomcat 7 +: <http://tecadmin.net/steps-to-install-tomcat-server-on-centos-rhel/>
* Windows 7/8/10
  + Java 1.8: <http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html>
  + Apache Maven 3.3 +: <https://maven.apache.org/install.html>
  + Apache Tomcat 7 +: <https://tomcat.apache.org/tomcat-7.0-doc/setup.html#Windows>

**Build Instructions**

* Extract contents of the zip file containing the project files.
* From the directory containing the extracted files, run the following command:   
  *mvn package*.
* Find the JAR file that has been created in the following directory:   
  *Backend\_Interactive\_Network\_Map/target*
* Move the JAR file from the target directory to the *Backend\_Interactive\_Network\_Map directory.*
* Run the backend server using the following command:  
  java -jar Backend\_Interactive\_Network\_Map-0.1.0.jar
* In the testLeaflet.js file, change the value of the localhost variable to the IP address and the port number in which the backend server is running. The file can be found in the following directory:   
  BuildingEquipmentMap/resources/js/testLeaflet.js
* Start the web client by running the test-page.html file on your favorite browser.

**Verification Plan**

In order to verify that the project installation is successful and the application is running, perform the following steps:

* Check the console running the JAR file for a message similar to the following:   
  2016-04-27 00:49:33.106 INFO 7264 --- [ main] backend.Application

: Started Application in 7.862 seconds

* Run the test-page.html page. The application should run smoothly without any lags and there should be no error messages on the browser console.